JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 339–364 (1999)

Research

ART: An Architectural Reverse Engineering Environment

ROBERTO FIUTEM[1†], GIULIO ANTONIOL[1], PAOLO TONELLA[1*] and ETTORE MERLO[2]
[1]*ITC-IRST, Istituto per la Ricerca Scientifica e Tecnologica, I-38050 Povo (Trento), Italy*
[2]*Department of Electrical and Computer Engineering, Ecole Polytechnique, C.P. 6079, Succ. Centre Ville, Montreal, Quebec, Canada*

SUMMARY

When programmers perform maintenance tasks, program understanding is often required. One of the first activities in understanding a software system is identifying its subsystems and their relations, i.e., its software architecture. Since a large part of the effort is spent in creating a mental model of the system under study, tools can help maintainers in managing the evolution of legacy systems by showing them architectural information.

This paper describes an environment for the architectural recovery of software systems called the architectural recovery tool (ART). The environment is based on a hierarchical architectural model that drives the application of a set of recognizers, each producing a different architectural view of a system or of some of its parts. Recognizers embody knowledge about *architectural clichés* and use flow analysis techniques to make their output more accurate.

To test the accuracy and effectiveness of the ART, a suite of public domain applications containing interesting architectural organizations was selected as a benchmark. Results are presented by showing ART performance in terms of precision and recall of the architectural concept retrieval process.

The results obtained show that cliché-based architectural recovery is feasible and the recovered information can be valuable support in reengineering and maintenance activities.

KEY WORDS: reverse architecturing; program understanding; cliché matching; flow analysis

1. INTRODUCTION

When the size and complexity of applications increase, the importance and criticality of the overall structure design, in other words, of the software architecture, increase. In the last five years, software architecture has attracted the attention of the software engineering community, with the seminal works of Garlan and Shaw (1996) and Perry and Wolf (1992). Much work has been published on module interconnection languages (Perry, 1987; Prieto-Diaz and Neighbors, 1986), architectural description languages (Abowd, Allen and Garlan, 1995; Allen and Garlan,

*Correspondence to: Paolo Tonella, ITC-IRST, Istituto per la Ricerca Scientifica e Tecnologica, I-38050 Povo (Trento), Italy. Email: tonella@itc.it
†At the time this work was performed R. Fiutem was with ITC-IRST. He is now with Sodalia S.p.A, 38100 Trento, Italy.

CCC 1040–550X/99/050339–26$17.50

Received 20 June 1998
Revised 15 July 1999

1994), *design patterns* (Coplien and Schmidt, 1995; Gamma *et al.*, 1995), and formal models for component integration (Luckham *et al.*, 1995; Ng and Kramer, 1995; Shaw *et al.*, 1995). The high diffusion of distributed, client–server systems and remote objects has pushed towards the development of interface description languages (Lamb, 1987), which allow one to control the exchange of structured data between different components of distributed communicating programs, and their implementation in industrial standards like CORBA (OMG, 1991) and DCOM.

Most of the research community's efforts have concentrated on providing methods and tools to design and formalize systems at the architectural level, thus supporting a *forward engineering* process. However, a *reverse engineering* approach that tries to extract descriptions at the architectural level from existing software artifacts may be extremely useful in maintenance activities. When approaching a legacy system, maintainers may want to know its overall organization and high level structure.

Other scenarios may require program understanding at the architectural level: for example, when the goal is to reuse some functionality of a system (Canfora *et al.*, 1994; Cimitile and Visaggio, 1995). Another motivation for architectural design recovery is to compare the architecture extracted from code with the intended design to detect discrepancy or to trace architectural evolution (sometimes called *architectural drift*) among different versions of a software product. In fact, two of the software evolution laws described in Lehman (1980) (Continuing Change and Increasing Complexity) state that if a software product is used then it will undergo continuing modifications and enhancements; this will worsen its quality, and hence also its architecture. Being able to monitor architectural evolution makes it possible to plan restructuring when the quality of the overall system structure has decreased below a safe or desired level.

For legacy systems, documentation is often poor and the people who originally developed the programs are no longer available, so the only source of documentation about the program is the source code itself. Since a great deal of effort is spent in creating a mental model of the system under study, tools can help maintainers in managing the evolution of legacy systems by showing them architectural information.

One of the approaches explored in the area of program understanding is that of trying to identify instances of *program concepts* at algorithmical and data-structure level using *plans*, which are abstract representations embodying the knowledge about program concepts, their components and constraints (Hartman, 1991; Kozaczynski, Ning and Engberts, 1992; Rich and Wills, 1990; Wills, 1992).

Other approaches have investigated structural analysis and *in-the-large* redocumentation of software systems, by building tools that support methods for identifying, reorganizing, and documenting layered subsystem hierarchies (Biggerstaff, 1989; Chen *et al.*, 1995; Holt and Pak, 1996; Wong *et al.*, 1995).

Recently, some works that address the problem of high-level design recovery using reverse engineering technology have been presented (Harris, Reubenstein and Yeh, 1995a; 1995b; Yeh, Harris and Chase, 1997). They integrate in a framework architectural style, representations and a library of recognizers to extract architectural information from source code. Such a framework, among other things, was used to reverse engineer the flow information between tasks in a legacy application (Holtzblatt *et al.*, 1997).

In this paper, an environment for architectural reverse engineering of software systems is described. The environment, called the architectural recovery tool (ART), is part of our program

understanding and reengineering environment CANTO (Code and Architecture aNalysis TOol) (Antoniol *et al.*, 1997), and supports the discovery of information at the architectural level, presenting such information to programmers and maintainers through multiple hierarchical views. It is based on a hierarchical architectural model that specifies component and connector types (Garlan and Shaw, 1996) of a software architecture at different levels of detail. The full description of the hierarchical model is presented in Section 2.

The architectural recovery process is based on architectural recognizers (Harris, Renbenstein and Yeh, 1995a) that work on abstract syntax trees (ASTs) extracted from source code. Architectural recognizers exploit *clichés* (Rich and Wills, 1990) to identify architectural constructs. To obtain more accurate results, flow analysis techniques are exploited to represent constraints among cliché components.

ART is intended for the C/Unix domain. It is built on top of the Refine/C[‡] and exploits its capabilities to build and analyze ASTs.

To test the accuracy and effectiveness of ART, a suite of public domain applications containing interesting architectural organizations was selected as a benchmark. Results are presented by showing ART performance in terms of architectural concepts retrieval precision and recall.

This paper is organized as follows. Section 2 illustrates the model that was used to represent architectural information and its implementation for the C/Unix platform. In Section 3, the cliché recognition techniques and the cliché library are presented. Cliché recognition problems are discussed with reference to the architectural concept recovery task. The set of recognizers available in the environment is also presented. Section 4 describes ART implementation, the experimental setup, the program test suite used to benchmark ART, and the results obtained on architectural concepts identification on the benchmark. Section 5 discusses related research, and Section 6 presents conclusions and future work.

2. ARCHITECTURAL MODEL

Several notations and languages have been proposed to describe software systems architecture (Allen and Garlan, 1994; Dean and Cordy, 1995; Inverardi and Wolf, 1995; Magee and Kramer, 1995; Prieto-Diaz and Neighbors, 1986; Shaw *et al.*, 1995). However, most of these notations have been used within forward engineering approaches. They are architectural or configuration/integration languages producing system descriptions that can be, in some cases, further processed to generate real software systems. These descriptions usually represent what is called the *idealized* (Harris, Reubenstein and Yeh, 1995b) or *conceptual* (Soni, Nord and Hofmeister, 1995) software architecture. From a reverse engineering point of view, it is difficult to recover such information because it is often distant from the actual objects that are retrievable from source code. Code represents the *as-built* (Harris, Reubenstein and Yeh, 1995b) architecture, that is the actual software organization. These two views do not necessarily match, and this is the bridge programmers have to mentally build when performing maintenance activity.

The model adopted for our architectural analysis environment starts from the *as-built* architecture and is based on partial views, each consisting of specific components and connectors, that represent different structural aspects of a software system.

[‡]Refine, Refine/C, Intervista and Workbench are trademarks of Reasoning Systems Inc.

These views are hierarchical in that components at one level of the hierarchy may be described by views of components and connectors at the lower level. The hierarchical nature of architectural information and the need for multiple levels of representation have also been stated in Ng and Kramer (1995) and Soni, Nord and Hofmeister (1995).

The highest abstraction level of our architectural model is the *system*, and the corresponding view is the *system view*. A system, $S = \langle P, I \rangle$, consists of a set P of *program* components that communicate through a set I of connectors of type *inter-process-connector* (IPC).

Programs are separately executable units that constitute an application and cooperate by exchanging data and synchronizing themselves through connectors.

The IPCs that can bind program components are represented in the following taxonomy, which is partially derived from Dean and Cordy (1995):

- **shared-file**: a data repository accessed sequentially by several programs.
- **shared-memory**: data that can be accessed at random and are shared among different programs.
- **remote procedure call (RPC)**: a procedure call-like mechanism between different programs.
- **stream**: a unidirectional or bidirectional, reliable and ordered connector between two programs.
- **message**: a connector that represents data exchange among an arbitrary number of programs.
- **invocation**: a connector that represents one program (process) starting another program (process).

The system view is particularly relevant for systems consisting of concurrent communication processes; applications consisting of a single program will be represented at the system level by a single component. A program component can, in turn, be represented by two views: the *module view* and the *task view*.

In the *module view*, a program $P = \langle M, U \rangle$ is represented as a set M of components of type *module*, and a set U of connectors of type *uses*. A module is an entity that provides a computational service (Ghezzi, Jazayen and Mandrioli, 1992) by exporting a well-defined functional or data abstraction, and which often corresponds to separately compilable units such as those provided in most programming languages. The connectors U represent *uses* (Ghezzi, Jazayen and Mandrioli, 1992) relations, and illustrate the import/export of resources among the modules of a program.

The *task view* of a program $P = \langle T, S \rangle$ consists of a set T of components of type *task*, and a set S of connectors of type *spawns*. A task or thread is a subprogram or a piece of code that can be executed concurrently, sharing the address space with other tasks within a program (see, for example, Deitel (1990)). A *spawns* connector links two tasks, of which the first is the creator (father) of the second task (child). The task view of a program represents possible multiple threads of control and their relations. This is a slightly different view with respect to that of the invocation connector in the system view. In fact, for the invocation connector the relation spans over executables (different processes), whereas for the *spawns* connector the relation spans over subprograms or pieces of code executed in parallel within the same process. The information represented in the two views can, however, overlap in some cases.

Each module in the module view can in turn be analyzed and represented as a set of components and connectors generating the *call-graph* and *data flow graph* views. In the former case, components are *subroutines* and its connectors represent control transfer from one subroutine to another; in the

Table 1. Summary of the architectural views with their components and connectors: the code view was replicated through all the levels because it can be applied to any level. The fourth and sixth columns, titled `C Implementation`, give respectively the implementation of components and connectors for the C/Unix environment

Level	View	Component	C Implementation	Connector	C Implementation
System	System View	Program	same	IPC connector	IPC (pipes, sockets . . .)
	Code View	Program, File Directory	same	Comprises, DependsOn	same
Program	Module View	Module	File	Uses	Funcall, Var. access
	Task View	Task	Process	Spawns	Fork
	Code View	Program, File, Directory	same	Comprises, DependsOn	same
Module	Call Graph	Subroutine	Function	Call	Function call
	Data Flow	Subroutine, Variable	Function, Variable	Data set/use	Set/Use/Ref/Deref
	Code View	Program, File, Directory	same	Comprises, DependsOn	same

latter, components are subroutines and data structures, and the view describes subroutine accesses to data structures.

Finally, the *code view* illustrates the source code organization in the development environment. The code view elements are: source files, directories, libraries, include files, and so on. At this level of abstraction possible relations are *is-component-of* and *depends-on*. The *is-component-of* relation indicates the files that are contained in directories, and in turn, the directories that compose the system. *Depends-on* relations represent compile dependences among files such as the ones that can be extracted from `makefiles` for example. The code view corresponds to what is called *code architecture* in Soni, Nord and Hofmeister (1995). This view is not tied to a particular level; rather it can be computed at all levels: system, program and module.

In order to apply and experiment our architectural recovery approach, this generic architectural model was instantiated in a model for a particular language/operating system. The C/Unix environment was chosen for our experimental platform because Unix and C are, respectively, a general-purpose and widely diffused operating system and language. Moreover, Unix has a rich variety of inter-process communication mechanisms, which allow systems to be implemented according to most of the architectural styles described in the literature (Garlan and Shaw, 1996). Other environments could be the object of experimentation, by defining the specific mappings between entities of the generic and the instantiated model.

In Table 1, the different views defined together with the associated components and connectors are summarized. Figure 1 shows the hierarchical relations among the defined views at the different levels. The code view is shown vertically as it can be applied at every level.

The fourth and sixth columns of Table 1 show the implementation of components and connectors at each level of the architectural model. The correspondence between generic components and connectors and their C/Unix instantiation is straightforward, except for *modules*. The C language does not provide explicit language constructs to define modules. A module could correspond to a group of files, a single file, a function, or a group of functions. Nevertheless, in C the visibility of definitions (functions, global variables or constants) can be controlled at the file level using the keyword `static` and the file inclusion mechanism. In the following, we assume that modules are

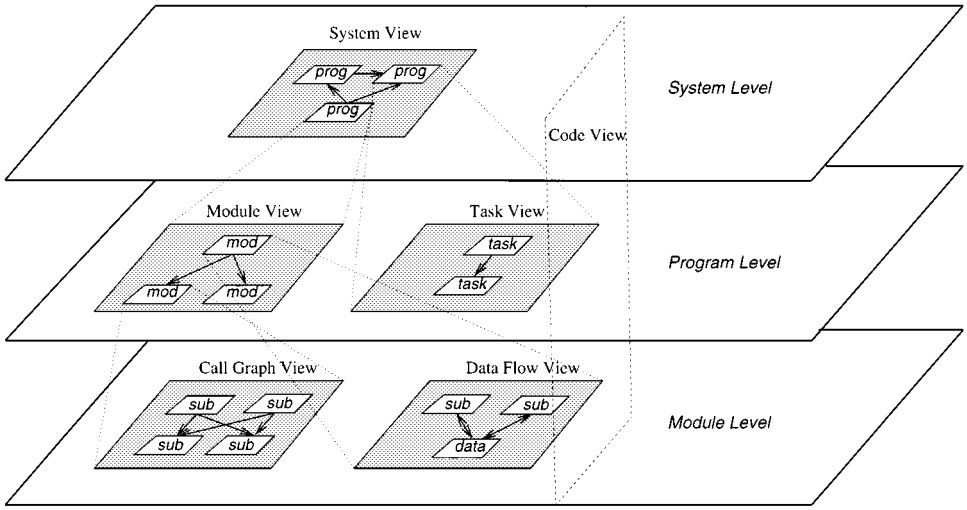

Figure 1. Hierarchical relations among the architectural views at the different levels

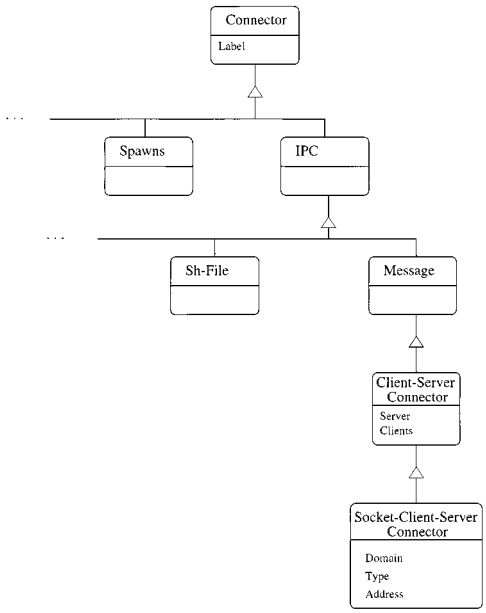

Figure 2. An example of connector hierarchy: a socket-based client–server connector for the C/Unix environment. Under each connector class name its attributes are listed; the arrows between boxes represent inheritance relations

implemented as separate files. In the near future, this assumption will be overcome by integrating in our recovery environment some modular abstraction recovery technique (Canfora *et al*., 1993; Girard and Koschke, 1997; Liu and Wilde, 1990; Yeh, Harris and Reubenstein, 1995) to produce better architectural descriptions.

Components and connectors may be specialized in several different implementations. For example, Figure 2 illustrates the relations between the general `client–server` connector and its C/Unix implementation based on sockets, using an object-oriented inheritance hierarchy. The same kind of connector could be implemented with other mechanisms such as pipes, RPC or others.

The proposed architectural model is mostly derived from those proposed in the literature. Our contribution lies in the hierarchical organization of the different types of components and connectors, in such a way that the software architecture of the system can be incrementally extracted by the subsequent application of different analyses at increasing levels of detail.

3. ARCHITECTURAL RECOVERY

3.1. Cliché representation

The process of understanding programs is usually based on recognizing aggregates of statements in the source which exhibit specific relations. These relations may be syntactic or may involve control and data flow. Such source code structures represent standard or *stereotypical* ways of implementing algorithmic computations or data structures (Wills, 1992). These aggregates are called *clichés* and represent a sort of knowledge about problem solving. Examples of clichés are list enumeration, binary search, or sorted list and priority queues. Clichés implementing more abstract structures may be composed of other clichés; the overall result of cliché recognition is called a *design tree* (Rich and Wills, 1990; Wills, 1992), i.e., a hierarchical description of a program's design.

A plan specifying a process generation cliché under the C/Unix environment is shown in Figure 3, expressed in a formalism similar to that proposed in Kozaczynski, Ning and Engbert's (1992). Here, components may be specified as sub-clichés or directly as patterns, using a Refine-like syntax, where '`@v`' and '`@@`' represent, respectively, a named and an unnamed single-value pattern variable, and '`$v`' and '`...`' represent respectively, a named and an unnamed multi-value variable. Constraints may enforce equivalence[§] among bound variables (AST subtrees) by using the same pattern variable name within component attributes. They may also enforce AST structural relations among components, e.g., `ancestor (A,B)` states that `B` `can` be reached from `A` along zero or more `parent` links. Figure 4 shows a code fragment whose AST representations can be matched by the *spawns* cliché of Figure 3: the AST fragments that correspond to the cliché components are shown in bold.

Beyond expressing purely syntactical relations, constraints may impose control/data flow relations among components. Essentially, the following additional interprocedural relations are used to specify control/data flow constraints (Kozaczynski, Ning and Engberts, 1992):

- `control-dep(A,B,C)`: A is control dependent on the branch of B that makes the C boolean condition true.

[§]Two subtrees are considered equivalent in this discussion if they have the same structure and all node attribute values are equal.


```
cliche spawns(CHILD: @child)
components
  S1: assign(LHS: @var1, RHS: "fork()")
  S2: test(VAR: @var2, C: @child, F: @father)
constraints
  data-dep(S2,S1,@var2)
```

Figure 3. Plan representation of a process generation cliché in the C/Unix environment

```
/* creation of a child process */
    pid = fork();
    if(pid == 0) {
        /* son's code */ }
    else {
        /* father code */ }
```

Figure 4. Example of a C code fragment whose AST can be matched by the spawn chichés

- `data-dep (A,B,D)`: the value of D used at A is defined by B.

Since they are interprocedural, such constraints overcome the problems of aliasing introduced by parameter passing and variable copy that affect a plan specified using only syntactical constraints. The benefits of adding control/data flow relations to the set of possible cliché constraints will be further discussed in Section 3.4.

3.2. Cliché library

The library of clichés embedded in ART has been built and instantiated for the C/Unix environment, and currently contains about 50 clichés. This number is growing since new clichés are currently being added to the library.

The architectural cliché library was acquired manually by analyzing several programming books, consulting experts, and also by directly reading source code examples. Although manual acquisition is expensive, for the architectural domain it only has to be done once as architectural concepts are rather independent of the specific application domain.

Clichés can be grossly classified into a hierarchy of levels according to both the nature of their components and the constraints used to bind their components. Figure 5 shows such layered organization with examples for each level referring to the `client` and `server` concepts.

The bottom level corresponds to the *syntactic* clichés, i.e., those which can be directly recognized by means of pattern matching on the AST.

The level above combines syntactic level clichés to form more abstract concepts, here called *abstract clichés*. An example is represented by the `server` cliché in Figure 6, which corresponds to the creation of a socket-based channel from the server side.

For these two levels, the kind of constraints used to recognize cliché instances are based on syntax and control/data flow. Recognition is performed on a single program; for this reason, clichés belonging to these two levels are classified as *intra-program* level clichés.

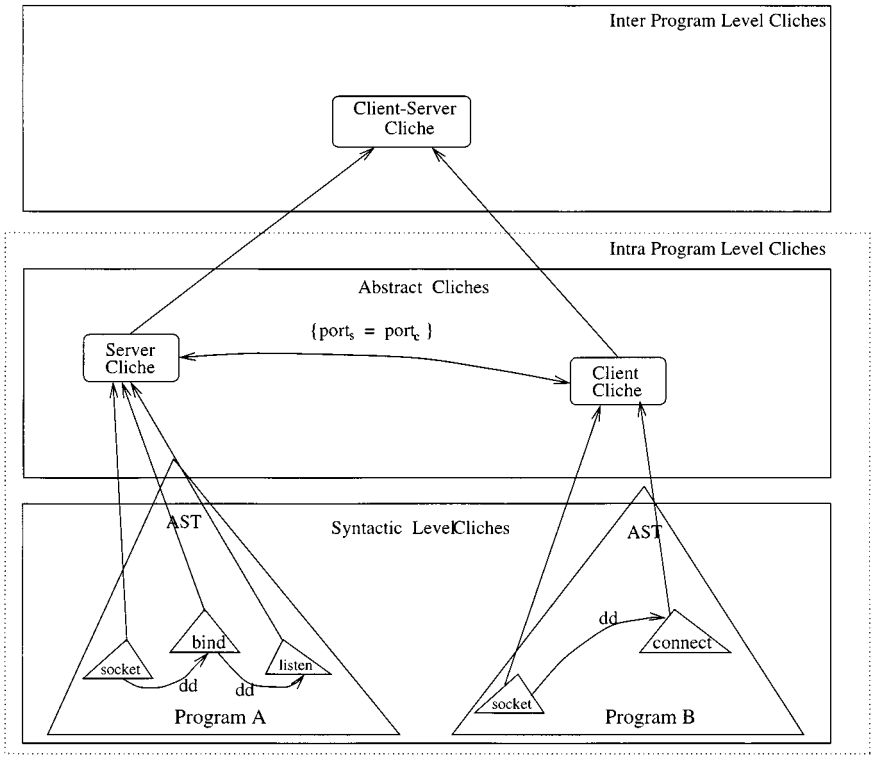

Figure 5. Layered organization of the cliché library

```
% syntactic level cliche'
cliche assign(LHS: @var, RHS: @value)
components
  S1: assignment-stmt(PATTERN: "@var = @value")

%  abstract cliche' implementing a client
cliche client(DOM: @d, TYPE: @t, PORT: @port)
components
  S1: socket-call(PATTERN: "socket(@d, @t, @@)")
  S2: assign(LHS: @s, RHS: @val)
  S3: connect-call(PATTERN: "connect(@r, @port)")
constraints
  ancestor(S1, @val)
  data-dep(S3, S2, @r)

% inter process communication cliche
cliche client-server
components
  S1: server(DOM: @d1, TYPE: @t1, PORT: @port1)
  S2: client(DOM: @d2, TYPE: @t2, PORT: @port2)
constraints
  @d1 = @d2
  @t1 = @t2
  @port1 = @port2

%  abstract cliche' implementing a server
cliche server(DOM: @d, TYPE: @t, PORT: @port)
components
  S1: socket-call(PATTERN: "socket(@d, @t, @@)")
  S2: assign(LHS: @s, RHS: @val)
  S3: bind-call(PATTERN: "bind(@r, @port)")
  S4: listen-call(PATTERN: "listen(@t)")
  S5: accept-call(PATTERN: "accept(@u)")
constraints
  ancestor(S1, @val)
  data-dep(S3, S2, @r)
  data-dep(S4, S2, @t)
  data-dep(S5, S2, @u)
```

Figure 6. An example of the clichés for each different level of Figure 5

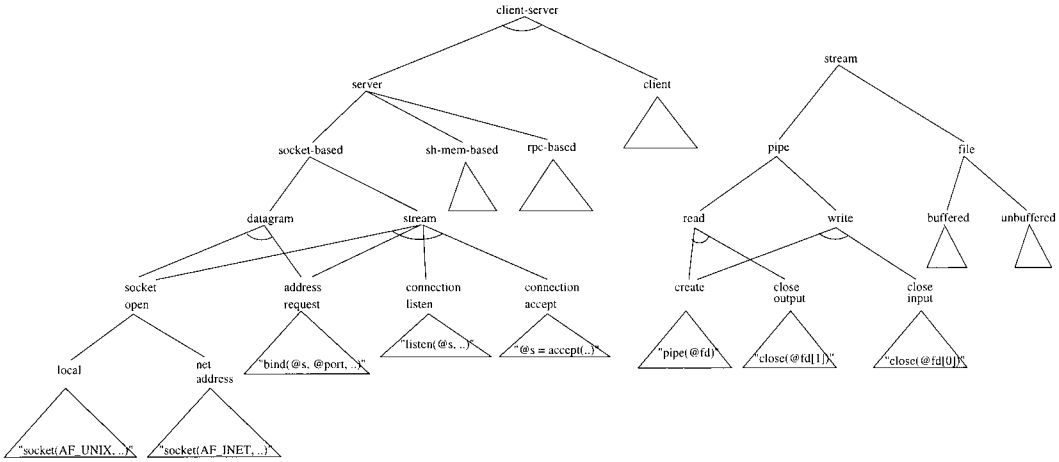

Figure 7. An excerpt of the ART cliché library

The next level of clichés in Figure 5 binds together abstract clichés by imposing different kinds of constraints with respect to the previous level. Cliché components belong to different programs and are bound together to find evidence of inter-process communication. This level is called the *inter-program* cliché level. The constraints imposed are based mainly on the equality of cliché attributes. An example is the `client–server` cliché, whose textual representation is reported in Figure 6. It corresponds to the creation of a communication channel between two different programs, one of which acts as a client and the other as a server. This kind of cliché generally expresses concepts such as communication among different processes: a precondition to recognize such concepts is to have one end of the communication channel in each program, and check the equality of specific attributes (the `DOM`, `TYPE` and `PORT` attributes in this case).

Figure 7 shows an excerpt from the library of clichés that have been codified in ART. The organization of the library is hierarchical as clichés are built hierarchically out of other clichés. The notation proposed in Wills (1992) was adopted for the library representation. There are two types of relations between the clichés in the library:

- *Composition*: Clichés may contain other clichés as components. Such a relation is represented in Figure 7 as a set of lines grouped by a circular arc from the composite to the component clichés.
- *Implementation*: A cliché may implement a more abstract cliché. Such a relation is represented by a line from the abstract cliché to its concrete implementations. For example, in Figure 7 a `socket-based` cliché is an implementation of the more general concept of `server`, while a `pipe` cliché is an implementation of the more general concept of `stream`.

Syntactic level clichés are represented as a pattern within a triangle to signify their placement in the AST. Subtrees that are not expanded are depicted as empty triangles.

3.3. Cliché recognition process

Cliché recognition is performed *bottom-up*, starting from the syntactic level clichés and applying constraints to them to find evidence of higher level clichés, following the hierarchy of Figure 5.

Before starting recognition, a preliminary analysis on source code is performed to eliminate *dead code*, i.e., the set of functions which are not reachable from the `main` in the call graph[¶]. This kind of analysis allows one to remove spurious clichés from the result that might be contained in the 'dead' functions. Removing dead code ensures that, at least statically, a path exists between the `main` and a function (or group of functions) containing a cliché. Moreover, dead code elimination is useful when source files are shared among different programs. In fact, when ART analyzes a distributed application made of several programs, it analyzes each program separately, so that if, for example, a utility module containing some clichés is shared among programs and therefore is multiply linked, such clichés are correctly attributed to each program according to the actual function calls.

The steps of the recognition process are as follows:

(i) Perform dead code elimination.
(ii) Find syntactic level clichés and store them (each program has a set of syntactic level clichés attached).
(iii) Find abstract clichés (e.g., `server`, `client`, `spawns`, etc.).
(iv) Find inter-program clichés, working on single program clichés.

So, for example, with reference to Figure 5, direct pattern matching on the ASTs of programs A and B (shown in the bottom part of the Figure), would identify the `socket-call`, `bind-call` and `listen-call` in one program and the `socket-call` and `connect-call` clichés in the other program. By applying the constraints specified in Figure 6, a `server` cliché would be matched in program A, and similarly, a `client` cliché in program B.

The cliché recognition process would then continue by checking the attributes of the `server` and `client` instances identified to find evidence of a `client`–`server` cliché, by imposing the equality of the corresponding attributes.

3.4. Cliché-related issues

An instance of a cliché may appear in several different forms due to variations. Wills (1992) classifies the main sources of variation as: *syntactic variation*, *implementation variation*, *delocalization*, *unrecognizable code*, *organization variation*, *redundancy*, and *function sharing*.

Syntactic variation mostly regards the syntactic level clichés as those which are recognized directly with pattern matching on the AST. The use of control/data flow based constraints makes the higher level clichés less dependent on the syntactic forms used.

There are many ways to achieve the same data and control flow, using different constructs of each different programming language. First of all, the chosen cliché representation abstracts away variable names as it uses syntactic pattern matching with placeholder. Then, the different cliché forms, corresponding to the different language constructs that may be used in specifying a cliché, are codified as different *implementations* (Wills, 1992) of the same concepts. A cliché recognizer

[¶]This step makes use of a conservative flow-insensitive, points-to analysis to also take into account the effect of calls through function pointers, adding a link to the functions potentially called.

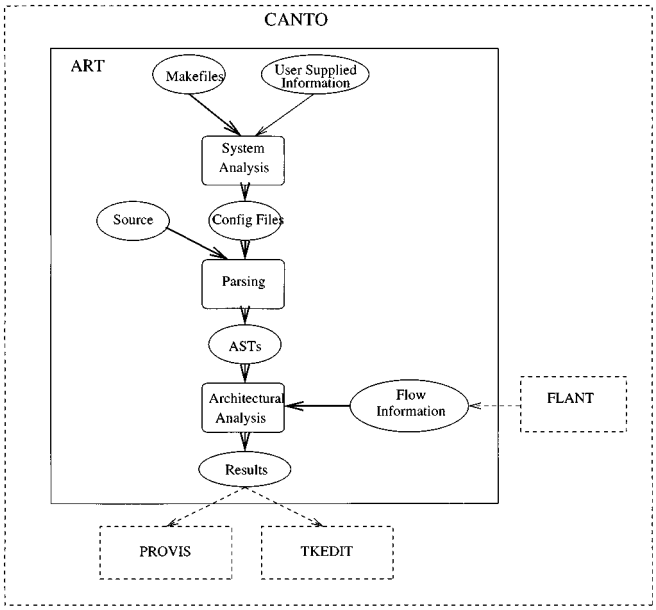

Figure 8. Organization of ART

embodies the knowledge of all the different forms that a certain cliché can assume. Moreover, in the architectural domain, the syntactic variability of bottom level clichés is not so high. The clichés that present the highest number of syntactic variants, and for which the highest number of correspondent patterns have been specified, are the `spawns` and `test-zero` clichés, which, respectively, require 4 and 5 different patterns to cover the different syntactic forms.

Implementation variation is related to the fact that a given concept may be implemented in different ways. The different cliché forms are codified as different implementations, i.e., a set of alternative forms, of the same concepts. The hierarchical organization of the cliché library allows the different implementations of clichés to be represented compactly. For example, as Figure 7 shows, different sub-clichés (`socket-based, rpc-based, sh-mem-based`, etc.) are used to recognize a `server` concept, corresponding to the different architectural mechanisms used under Unix (sockets, remote procedure calls, and shared memory, respectively).

Delocalization (Letowsky and Soloway, 1986; Rugaber, Stirewalt and Wills, 1995; Wills, 1992) and *organization variation* are quite similar issues. They occur when the components of a given cliché are not contiguous, but spread over different procedures or even different files in the source code. This phenomenon causes some problems to a purely syntactically based cliché recognition. However, by exploiting inter-procedural control and data flow relations, the delocalization problem can be overcome for many of the clichés.

The *redundancy* problem occurs when part of a cliché appears more than once in the same instance of a cliché, or value propagation among variables is present. Again, by exploiting control and data flow based constraints, the problem of value propagation can be solved, since this kind

of constraint allows relations to be expressed directly on the cliché components, by-passing the redundant code.

The *unrecognizable code* phenomenon is also called *partial recognition*. The recognition system, because of the incompleteness of its cliché library, is not able to classify every line of the source code as belonging to some cliché. What actually happens is that a forest of design trees is generated, and some parts of the program remain uncovered. For the architectural domain, partial recognition of a program is expected, since most of a program is usually devoted to performing the computation related to its application domain, while only a small fraction of the code is devoted to implementing architectural structures. Partial recognition is not a problem for our recognition technique, as it is a bottom-up approach and does not require, as in Wills (1992), that the clichés in the library are applied by a parser as grammar rules to an entire program representation. There is, all the same, the problem of completeness of the library with respect to the possible clichés that can be used to implement architectural structures.

Function sharing is also called *optimization* (Wills, 1992), or *overlapping implementations* (Rich and Wills, 1990). It happens when the same portion of code is shared among two or more clichés. Here the implementations of the clichés overlap. This phenomenon does not represent an actual problem given the hierarchical organization of the cliché library. This simply means that in the design tree resulting from the analysis of a program, some cliché components will be repeated, or, if the repeated objects are factorized, that the design tree will actually become a directed acyclic graph.

4. EXPERIMENTAL SETUP AND RESULTS

4.1. Analysis environment

The architectural model, cliché recognition techniques, and the recognizers described in the previous sections were implemented in ART. ART is based on Refine and exploits the results of the flow analyses performed by FLANT (FLow ANalysis Tool), which is also part of CANTO.

The organization of ART is shown in Figure 8. The first phase of the analysis process supported by ART is called system analysis, and determines the components at system level, i.e., the separate independent programs that are part of a system. This phase is only partially automated by a `makefile` analysis. If different build mechanisms are used, the user can supply a system description file. Once the C programs involved in the system under analysis have been identified and written in the corresponding configuration files, the Refine/C package is used in the Parsing phase to parse them and produce the corresponding abstract syntax trees (ASTs).

All architectural recognizers work on the ASTs produced by the Refine/C parser. Several recognizers are applicable to the ASTs, and each of them is related to a specific level of the architectural model. They produce the different architectural descriptions of the source code. Beyond the information in the ASTs, the recognizers that employ cliché recognition also need to check control/data flow based constraints. For this purpose, ART is connected to FLANT through a socket-based communication channel and can issue queries to FLANT about the satisfaction of flow based constraints.

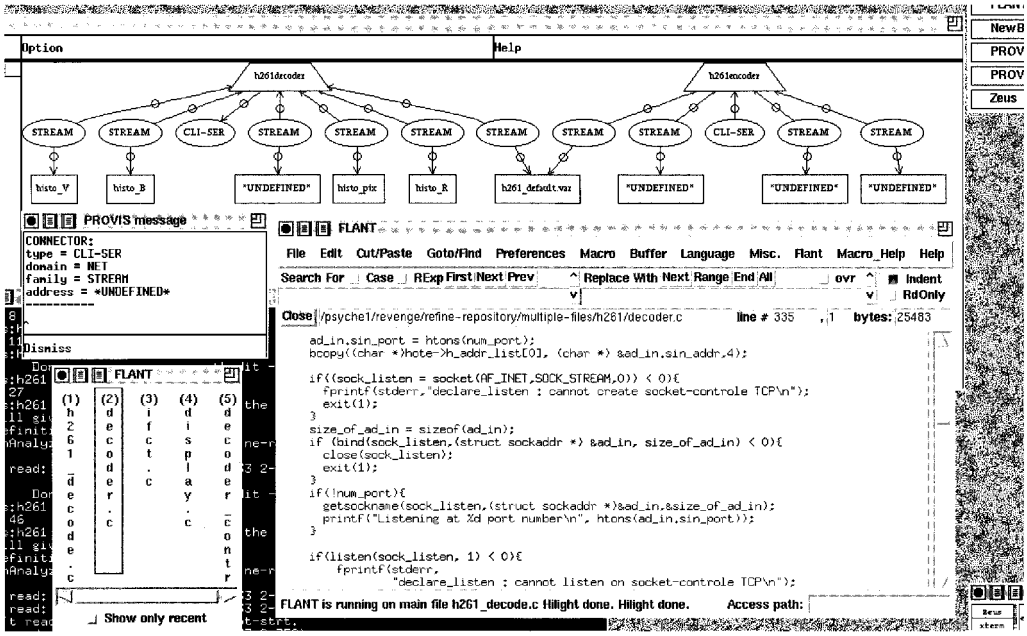

Figure 9. Example of interaction with CANTO

ART produces results mainly in terms of graphs of components and connectors, representing the different architectural views extracted by each recognizer. Such results are shown graphically by PROVIS, the graph browsing module of CANTO.

ART also produces a list of all the clichés matched during cliché recognition, together with the list of the actual statements in the source code that caused recognition. Such information can be shown by PROVIS in the form of a design graph, or by text highlighting with TKEDIT, the customized text editor, which is also part of CANTO.

Figure 9 shows the results of an interaction with ART in an example application. The top window, displayed by PROVIS, is the system view of the application, while the bottom right window, displayed by TKEDIT, highlights the lines of code that implement a `client–server` connector.

4.2. Experiment

The program test suite used to benchmark ART performance consists of four public domain applications written for the C/Unix platform. Using public domain programs has some advantages: source code can be easily obtained and results can be compared with those of related works. Table 2 lists the applications that were analyzed and their total size in lines of code (LOC), including header files. When an application contains multiple programs, beyond showing the total size of the source code, the size of each program was also given.

The application size is fairly spread between small sized applications like `H261` and medium sized applications like `Mosaic`.

Table 2. Program test suite for ART benchmarking: size is given in lines of code

Application	Total LOC	Programs	Program LOC
H261	6923	h261encoder	1977
		h261decoder	2004
Samba (v1.9.13)	31537	nmbd	6618
		smbclient	8884
		smbrun	80
		smbd	22463
		smbstatus	14074
Bash (v1.14.7)	37575	bash	37575
Mosaic (v2.7b4)	100135	mosaic	100135
Total	176170		

The `H261` software tool implements the H261 standard, a widely diffused image compression standard for teleconferencing defined by CCITT. The `h261encoder` program encodes incoming data and sends them over the network, and the `h261decoder` program reads data from the network, decodes them, and displays the decoded image in an X window.

`Samba` is a well-known software package that allows to access PC resources (disks and printers) from Unix, Netware and other operating systems. The package consists of several programs communicating over the network using TCP/IP.

`Bash` is the GNU Project's Bourne Again SHell, an interactive shell derived from the Bourne shell. Since it is a shell, it contains interesting architectural mechanisms for process control and communication.

`Mosaic` is the well-known forerunner of all World Wide Web browsers, developed at the National Center for Supercomputing Applications at the University of Illinois.

Each application of the test suite was loaded in the ART environment and all the architectural recognizers available were applied. In the following, the results related to the system level recognizers plus the `SpawnsRec` recognizer are discussed, since it is more interesting to show the effectiveness of cliché based recognition; the remaining recognizers are not based on cliché matching, and hence their results were not included in the discussion.

4.3. Results

Results of architectural analysis are given here in terms of identified architectural concepts over the program test suite.

Since it was not possible to consult the designers of the analyzed applications, to verify the accurateness of the ART output the identified architectural concepts were checked manually by the authors directly on the source code. Given the relatively small cardinality of the set of concepts identified, this was not a heavy task: verification for the whole test suite required about 5 hours. Note that such work is exactly what a software engineer would do, starting from the results of ART. ART suggests hints to the user: the source code starting points on which he/she may concentrate to understand the software architecture.

Table 3. Results of architectural recovery on the `H261` system

	h261encoder	h261decoder	H261
Client	1	0	
Server	0	1	
File	4	6	
Spawns	1	5	
F2CPipe	1	4	
Client–Server			1
SharedFile			1
Total time (s)		799	
Recognition Time (s)		236	
Precision (%)	100	100	100
Recall (%)	100	100	100

To benchmark ART performance, the most commonly used measure of retrieval effectiveness was used, which is *recall* and *precision* (Frakes and Baeza-Yates, 1992). In fact, the ART recovery process can be seen as the retrieval, or even classification, of architectural concepts present in the source code.

Given a query in a 'database', *recall* and *precision* are defined as:

$$recall = \frac{\#\ relevant\ documents\ retrieved}{\#\ relevant\ documents} \tag{1}$$

$$precision = \frac{\#\ relevant\ documents\ retrieved}{\#\ documents\ retrieved} \tag{2}$$

The fewer the *false negatives* present in the results, the higher the system recall. The fewer the *false positives* present in the results, the higher the system precision.

Results will be given in terms of abstract clichés, leaving out syntactic clichés; most of the time, these are too numerous and too low-level to be interesting for the software engineer analyzing the application.

Table 3 reports the architectural concepts recovered for the `H261` system. Among all the concepts that are known to ART, only those for which some instances were actually found in the source code are reported. The first five rows report results about intra program level clichés: `Client, Server, File, Spawns` and `F2CPipe` (the `F2CPipe` cliché represents a father process opening a pipe to read/write data to/from a child). Results are reported singularly for each of the two programs that are part of `H261`. The fourth and sixth rows report results over the inter program level clichés identified, the `Client–Server` and `SharedFile` clichés. For each cliché, the number of instances retrieved by the ART is reported.

By simply reading this information, a software engineer can obtain useful information about the overall structure of the application. In fact, the presence of a `Client–Server` cliché at the application level shows us that the two programs communicate over a TCP/IP socket channel. This could also be guessed by the presence of a `client` and `server` cliché, respectively, in the `h261encoder` and `h261decoder` program. As a consequence, it is also known which is the server (`h261decoder`) and which is the client (`h261encoder`).

Table 4. Results of architectural recovery on the Samba system

	nmbd	smbclient	smbd	smbstatus	smbrun	Samba
Client	0	1	0	0	0	
Server	1	0	1	0	0	
File	4	3	16	9	0	
Spawns	1	0	1	0	0	
Invocation	0	1	1	0	1	
Client–Server						1
SharedFile						3
Total time (s)				3463		
Recognition Time (s)				207		
Precision (%)	100	100	100	100	100	25
Recall (%)	100	100	100	100	100	100

A SharedFile cliché was also identified: this means that there is a file opened by both programs. This can represent a further communication channel, beyond the Client–Server one.

Then the presence of a Spawns clichés in both the h261encoder and h261decoder programs indicates that such programs create new processes. This suggests the adoption of a classical structure, in which a daemon process generates as many servers as required by clients, forking new processes. Note here that the number of instances actually identified by the ART corresponds to those statically determinable. For example, five Spawns clichés were identified for the h261decoder program, and actually five calls to the fork system call—the Unix process generation system call—are present in source code. However, by manually analyzing the source code one can realize that some of these calls are mutually exclusive; probably the actual process instances created dynamically by the program are less than those statically retrieved. This information, however, cannot be completely determined by a static analysis.

The last four rows of the table give, respectively, the total execution time needed to analyze each entire application on a SparcStation 10, the net recognition time, and the average precision and recall values for each program and for the entire application. The total execution time includes the time for loading the ASTs and the time waiting for FLANT answers on the flow constraints evaluation. Such times range, respectively, from 15% to 64% and from 5% to 70% of the total time on the test suite. Average precision values and recall values are computed over all clichés. Performance is very good on this system: all cliché instances present in the source code were retrieved by the ART. Neither false positives nor false negatives are present, giving 100% precision and recall for both programs.

The analysis of the Samba system reveals some interesting information about the architectural organization of the five programs that compose the application. It turns out that smbd opens a communication channel from the server side, while the smbclient opens a channel on the client side. The port attribute of both such channels may be provided by the user or may correspond to a default value, which is the same for both programs (the value 139). A Client–Server cliché among the two programs is therefore identified based on port value matching. The nmbd program also contains a Server cliché, which, by the static analysis of attributes is of type DATA_GRAM. This is a connectionless communication mechanism, on a user-provided port number, which defaults to 137. External processes knowing such a service number may send or receive single message

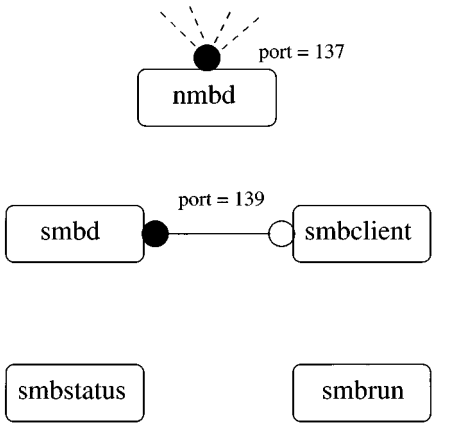

Figure 10. View of the communication channels in the `Samba` *system as recovered by ART*

packets without an explicit connection. Figure 10 graphically illustrates the information recovered about inter-process communication. Small circles represent communication end-points; filled circles correspond to servers while empty circles correspond to clients (following the conventions adopted in Darwin (Ng and Kramer, 1995)).

Both `nmbd` and `smbd` programs contain a `Spawns` cliché. This indicates that they may be run like daemons (forking a child process and detaching from the terminal).

As regards inter program cliché recognition, beyond the already mentioned `Client–Server` cliché, three spurious `SharedFile` clichés were identified, which lower the precision at the application level to 25%. Such errors are due to the incidental matching of filenames for some `File` clichés identified in different programs. In these cases, multiple values are recovered as possible values for the `filename` attribute, among which are many spurious values. Such spurious values have essentially two causes: the first is the use of data dependence closure in determining the statements which may potentially define the attribute value. By transitively following use-definition chains, values related to variables that only indirectly affect the variable of interest may be included. The second reason lies in the imprecision with which attribute values are computed.

In fact, all constants involved in expressions that are in the transitive closure are reported as possible values, with no further analysis. Since recognition of `SharedFile` clichés is based on `filename` attribute matching, spurious cliché instances are recognized.

Since no false negatives were reported during recognition, recall again corresponds to 100%.

Since `bash` is an interactive shell, a command interpreter, it is expected that invocation of external executables, internal generation of children processes and pipes are used. In fact, such concepts are all identified by the ART at several points in the program.

The two points at which external commands are invoked were identified as `Invocation` clichés, and the other two points at which external process generation code is present (`Spawns` clichés) were also identified. Some piping mechanisms were identified. However, in this case, two of the seven `Pipe` clichés identified are false positives. This is because the `Pipe` cliché checks whether a pipe is created and a `close` operation is made at one of the two ends of the pipe for the purpose of setting the direction of data flow through the pipe. For these two false positives, the

semantics of the `close` operation is not to set the direction; rather, since they are contained in an exception handling section of the code, to close all open descriptors and return an error. These two false positives lower the overall precision to 83.3%.

Four `F2CPipe` clichés are actually present in the code, associated with four points at which the main process generates a child process and opens a communication channel based on pipes to talk with the child. Such clichés were not identified by the ART, because these instances of the cliché do not satisfy the constraints imposed. In fact, it is required that one of the two `pipe` components of the `father-to-child-pipe` is contained in the code recognized as part of the `spawns` component. However, since the `spawns` is contained in a function different from that containing the two `pipe` components, the ART was not able to classify that code as part of the `spawns`. This is a case in which the delocalization problem prevents recognition.

These four false negatives have an impact on the overall recall which is 71.4%.

The largest program in the test suite is `Mosaic`. Its analysis revealed `Client` and `Server` cliché instances used to accept CCI connections and interact with Internet services (HTTP, FTP, GOPHER, etc.). `Invocation` and `Spawns` cliché instances correspond to the creation of external processes in response to browsing actions, such as clicking on an anchor representing a Postscript document. More detailed information about `Mosaic` analysis can be found in Tonella *et al.* (1996). Although the program is medium sized, the execution time, which corresponds to 5100 sec, remains acceptable in the authors' opinion with respect to the common needs of the users of this analysis.

The precision is 100% for almost all the clichés, except for the `Server`, in which case one of the three instances recognized is spurious, lowering the overall precision to 92.3%. The spurious instance is caused by overlapping implementations.

Beyond identifying architectural cliché instances, which represent connectors and components, the ART is able to retrieve attributes of recognized connectors and components. To do this, a variance of *copy constant propagation* (Aho, Sethi and Ullman, 1985) is used. Starting from the identified cliché, the ART computes the closure of the data dependences on the AST node that correspond to the attribute whose value has to be retrieved. For each of the statements in the data dependence closure, the ART looks for constants of a type that is compatible with the attribute type.

Table 5 reports the results of attribute values retrieved by exploiting data dependence closure on the program test suite for the `port` attribute of the `Client` and `Server` clichés, the `filename` attribute for the `File` cliché (not computed on `Mosaic`), and the `cmd` attribute for the `Invocation` cliché. For each application in the test suite, and for each attribute to be retrieved, columns labelled with 1 report the number of cliché instances for which the attribute value of interest could be determined exactly, resulting in only one constant value. The column labelled with >1 corresponds to cliché instances whose attributes have multiple constant values (due to multiple possible flows of control). Finally, the NC column reports the number of cliché instances whose attribute value may be non-constant. In such cases, the value is not statically determinable because it is provided at run-time by the user or read from a file. Note that the three categories are not completely disjoint: in particular, an attribute value might be a single value or a set of values, but also non-constant. This is because there could be places in the code where the value of this attribute is set to an explicit constant, and other places in which the same attribute value is run-time specified via a user interaction.

Note that most of the time, the attribute values are non constant, indicating that often information about which port values to use in inter-process communication, which command to execute, or

Table 5. Results on retrieval of values for some of the cliché attributes on the program test suite

	port cardinality			filename cardinality			cmd cardinality		
Application	1	>1	NC	1	>1	NC	1	>1	NC
H261	0	0	1	6	3	4	0	0	0
Samba	2	0	3	6	0	24	0	0	3
Bash	0	0	0	0	0	1	0	0	2
Mosaic	1	1	2	–	–	–	1	1	4
Total	3	1	7	13	3	30	1	1	0

which file to open, is provided by the user or read by external files, rather than being coded in the source code. This supports the finding that some part of the architectural information about a system is often not statically determinable.

5. RELATED WORK

The work presented in this paper is a practical attempt to apply methods and techniques from the program comprehension research area to the problem of understanding the architecture of software systems.

The program comprehension research area was greatly influenced by the *Programmer's Apprentice* project (Rich and Waters, 1988). In Rich and Wills (1990), programming clichés are represented using a flow graph based representation scheme. GRASPR (Wills, 1992) uses an efficient chart based graph parsing algorithm for clichés recognition. Information about the program's control flow, recursion and data aggregation is captured by adding specific attributes in the flow graph representation. Such attributes are taken into consideration by the recognition process. Some approaches toward recognizing aliasing and side effects to mutable data structures were also introduced. The target language was common Lisp, and the approach was tested on real-world, medium-sized Lisp programs in the domain of discrete-event sequential simulators.

A graph-based representation was also used by UNPROG (Hartman, 1991) to hierarchically combine control and data flow graphs into a hierarchical program model (HMODEL). An efficient matcher (HMATCH) was developed to recognize standard implementation plans (STIMPs) with the different HMODELs corresponding to a system. The approach is language-independent and was tested on unstructured programs written in various imperative languages, including COBOL, C, Pascal, PL/1, Lisp, and pseudo-code.

Other approaches take different perspectives: the DESIRE system (Biggertaff, 1989) uses *informal information* such as keywords, comments, identifier names and design documents to search for abstract concepts. PROUST (Johnson and Soloway, 1985) and DUDU (Allermang, 1990) use a top-down strategy of analysis. They require the user to specify top-level functional goals and check whether and how these goals are implemented in the code. However, they cannot analyze arbitrary programs to derive the goals being implemented by a generic program.

In Kozacynski, Ning and Engberts (1992), a language independent approach for recognizing abstract concepts in source code was presented. Concepts were classified into programming concepts, architectural concepts, and domain concepts. Semantic information, in terms of the

control flow and data dependence relationship among abstract syntax tree (AST) objects, was precomputed and represented as attributes in the AST. Data dependences were computed as a result of reaching definitions analysis obtained by a fixpoint iterative method. The focus of the research was concept recognition and transformation, and the system was experimentally validated on 8000 COBOL programs, ranging in size from 2000 to 15000 LOC, from a commercial production control application.

Rigi (Wong *et al.*, 1995) focuses on the architectural aspects of program understanding: it supports a method for identifying, building and documenting layered subsystem hierarchies. A powerful, customizable user interface allows effective human intervention in the recovery of higher level abstractions. Rigi was successfully applied to several real-world software systems, including C, Cobol and PL/AS programs with a size of up to two million LOC.

A framework for architectural recovery built on top of a Refine-based source code analysis environment, and targeted for the C/Unix environment, together with methods to manipulate software architecture views and some applications were presented in Harris, Reubenstein and Yeh (1995a; 1995b), Holtzblatt *et al.* (1997) and Yeh, Harris and Chase (1997). Recognition rules operate on source code level ASTs and collect architectural component and connector instances. The approach used in this work is different from the classical cliché matching approach of program understanding: recognition rules do not require algorithmic equivalence of a plan and source being matched; rather they are based on source code local events (e.g., system calls).

The work presented in this paper was inspired by the above mentioned research results. In particular, similarly to Kozacynski, Ning and Engberts (1992) and in contrast with Rich and Wills (1990) and Hartman (1991), we adopted the approach of a Refine-based AST annotated with flow information obtained by an iterative fixpoint method. Our approach aims also to take advantage of recent developments in software architectures (Garlan and Shaw, 1996) (namely the definition of architectural styles and abstractions), by defining specific architectural clichés that reflect these newly discovered abstractions. Although the approach presented is language-independent, the target language for the experimental part is C.

Similarities between our work and that of Harris, Reubenstein and Yeh (1995b) can be identified, mainly as regards the overall goals and the approach used, in particular in the way recognizers are applied. However, in their work there is no explicit use of clichés; rather, recognizers are written in a procedural language (built upon the Refine programming language), and the use of flow information is limited to some special cases (e.g., to retrieve values of system call parameters set by other sections of code).

With respect to Rigi, while the kind of architectural components and connectors it handles are concentrated on standard imperative language constructs (such as subprogram calls), and hence the resulting analyses deal mainly with identification of subsystems of interest within calling hierarchies, our recognizers deal both with intra-program architectural structures and inter-program distributed aspects like inter-process communication or dynamic process generation.

As regards the use of flow analysis, advances with respect to previous work were made to take into consideration C-specific language features such as pointers, function pointers and parameter induced aliasing. To achieve scalability to large industrial size systems, special attention was also devoted to the speed of convergence of the iterative fixpoint method by conceiving a flexible analyzer (Tonella *et al.*, 1997) that allows fine tuning of the trade-off between execution time performance and flow information precision.

6. CONCLUSIONS

Maintenance of large software systems may benefit from the support of reverse engineering tools, able to extract architectural descriptions from existing software artifacts. The lack, or inadequacy, of existing documentation is one of the main reasons for recovering architectural information directly from source code.

This paper presented the ART, an environment for architectural reverse engineering targeted at the C/Unix domain, working on abstract syntax trees and exploiting the knowledge of clichés to find instances of architectural concepts. It was found that a cliché-based recognition technique is feasible for architectural recovery. The number of clichés to be provided in the library is reasonable: with about 50 different clichés we were able to build recognizers for six architectural styles. Unlike classical plan recognition at algorithmic and data structure level, which tries to explain an entire program in terms of known clichés, here partial matches are allowed. Moreover, partial matches are expected, since the code devoted to implementing architectural structures is usually a small fraction of the total.

Exploiting flow analysis techniques to represent constraints among cliché component's allows one, in most cases, to specify clichés in a more general form, which is not affected by many of the problems of cliché-based recognition techniques such as delocalization, organization variation and redundancy.

A problem which remains open is that of the multiplicity of recognized clichés: how many instances of a given cliché contained in a given function are actually created? A static analysis cannot, in general, answer this question. To face this problem, at least partially, the knowledge of the call graph to preliminarily eliminate dead code from the sources to be analyzed is used. This guarantees that the clichés contained in a function, or group of functions, are potentially called under some conditions on input data.

The accuracy and effectiveness of the ART were tested on a suite of public domain applications ranging from small to medium-sized programs, for a total size of about 180 KLOC. The average execution time is about 70 sec per KLOC: this figure is reasonable, considering that architectural information is usually more stable than fine-grained information about data-structures and algorithms; it can thus be computed once and used until some major revisions of the software occur.

Recognition performance was good on the test suite: recall was 100% in three of four applications of the test suite, meaning that the number of false negatives is usually very low. Precision was greater than 80% for all programs analyzed, and dropped at 25% in only one case over four, and only at the inter-program level. Such performance figures indicate that cliché-based architectural recovery may be a valuable technique to support programmers in the task of understanding programs.

The cliché-based recognition related problems analyzed in Section 3.4 had little impact on the experimental results. Only delocalization and function sharing, respectively, caused misrecognition in `Bash`, in which it gave origin to four false negatives, and in `Mosaic`, in which a false positive was generated.

The static computation of cliché attributes affects, in some cases, the quality of inter-program level cliché recognition, which exploits information about attribute values. In fact, the use of data dependence closure may introduce spurious values, which in turn may generate spurious connections between components. Some heuristics about attribute value ranges (e.g., user defined socket ports must be greater than 1024) may improve such outcomings.

In addition, in the test suite clichés attributes were found in many cases to be non-constant, meaning that often such information is provided by the user or read by external files, rather than being coded in the source. This prevented recognition of some inter-program level clichés, which a manual analysis revealed to be present. Such a limitation is intrinsic to static analysis. It may be partially overcome by considering not only source code as an input of the reverse engineering process, but also other artifacts such as shell scripts, configuration files, build files, etc.

Future work will be devoted to a further experimental evaluation of our approach. We are also working at augmenting the library of architectural recognizers to enlarge the number of styles we are able to recognize.

References

Abowd G, Allen R, Garlan D. 1995. Formalizing style to understand descriptions of software architecture, *Technical Report* CMU-CS-95-111, School of Computer Science, Carnegie Mellon University, Pittsburgh PA.

Aho AV, Sethi R, Ullman JD. 1985. *Compilers: Principles, Techniques, and Tools*; Addison-Wesley Publishing Co.: Reading MA.

Allemang D. 1990. Understanding programs as devices, *Ph.D Dissertation*, Ohio State University, Columbus OH.

Allen R, Garlan D. 1994. Formalizing architectural connection, *Proceedings of the International Conference on Software Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 71–80.

Antoniol G, Fiutem R, Lutteri G, Tonella P, Zanfei S. 1997. Program understanding and maintenance with the CANTO environment, *Proceedings of the International Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; pp. 72–81.

Biggerstaff T. 1989. Design recovery for maintenance and reuse. *IEEE Computer* **22**(7):36–49.

Canfora G, Cimitile A, Munro M, Taylor CJ. 1993. Extracting abstract data type from C programs: A case study. *Proceedings of the International Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; pp. 200–209.

Canfora G, Cimitile A, Tortorella M, Munro M. 1994. A precise method for identifying reusable abstract data types in code. *Proceedings of the International Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; pp. 404–413.

Chen YR, Flower GS, Koutsofios E, Wallach RS. 1995. Ciao: A graphical navigator for software document repositories. *Proceedings of the International Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; pp. 66–75.

Cimitile A, Visaggio G. 1995. Software salvaging and the call dominance tree. *Journal of Systems and Software* **28**:117–127.

Coplien JO, Schmidt DC (eds). 1995. *Pattern Languages of Program Design*; Addison-Wesley Publishing Co.: Reading MA.

Dean TR, Cordy JR. 1995. A syntactic theory of software architecture. *IEEE Transactions on Software Engineering* **21**(4):302–313.

Deitel HM. 1990. *An Introduction to Operating Systems*; Addison-Wesley Publishing Co.: Reading MA.

Frakes WB, Baeza-Yates R. 1992. *Information Retrieval: Data Structures and Algorithms*; Prentice-Hall: Englewood Cliffs NJ.

Gamma E, Helm R, Johnson R, Vlissides J. 1995. *Design Patterns: Elements of Reusable Object Oriented Software*; Addison-Wesley Publishing Co.: Reading MA.

Garlan D, Shaw M. 1996. *Software Architecture: Perspectives on an Emerging Discipline*, vol. 1; Prentice-Hall: Englewood Cliffs NJ.

Ghezzi C, Jazayeri M, Mandrioli D. 1992. *Fundamentals of Software Engineering*; Prentice-Hall: Englewood Cliffs NJ.

Girard JF, Koschke R. 1997. Finding components in a hierarchy of modules: a step towards architectural understanding. *Proceedings of the International Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; pp. 72–81.

Harris DR, Reubenstein HB, Yeh AS. 1995a. Recognizers for extracting architectural features from source code. *Proceedings of the Working Conference on Reverse Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 252–261.

Harris DR, Reubenstein HB, Yeh AS. 1995b. Reverse engineering to the architectural level. *Proceedings of the International Conference on Software Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 186–195.

Hartman J. 1991. Automatic control understanding for natural programs. *Ph.D Dissertation*, Department of Computer Sciences, University of Texas, Austin TX.

Holt R, Pak JY. 1996. Gase: Visualizing software evolution-in-the-large. *Proceedings of the Working Conference on Reverse Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 163–166.

Holtzblatt LJ, Piazza RL, Reubenstein HB, Roberts SN, Harris DH. 1997. Design recovery for distributed systems. *IEEE Transactions on Software Engineering* **23**(7):461–472.

Inverardi P, Wolf AL. 1995. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering* **21**(4):373–386.

Johnson WL, Soloway E. 1985. Proust: knowledge-based program understanding. *IEEE Transactions on Software Engineering* **11**(3):267–275.

Kozaczynski V, Ning JQ, Engberts A. 1992. Program concept recognition and transformation. *IEEE Transactions on Software Engineering* **18**(12):1065–1075.

Lamb DA. 1987. Idl: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems* **9**(3):297–318.

Lehman MM. 1980. Programs, life cycles and laws of software evolution. *Proceedings of the IEEE* **68**(9):1060–1076.

Letowsky S, Soloway E. 1986. Delocalized plans and program comprehension. *IEEE Software* **3**(3):41–49.

Liu S, Wilde N. 1990. Identifying objects in a conventional procedural language: An example of data design recovery. *Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; pp. 266–271.

Luckham DC, Kenney JJ, Augustin LM, Vera J, Bryan D, Mann W. 1995. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering* **21**(4):336–355.

Magee J, Kramer J. 1995. Modeling distributed software architectures. *First International Workshop on Architectures for Software Systems*; IEEE Computer Society Press: Los Alamitos CA; pp. 206–221.

Ng K, Kramer J. 1995. Automated support for distributed software design. *Proceedings of 7th International Workshop on Computer-Aided Software Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 381–390.

OMG. 1991. The Common Object Request Broker: Architecture and specification. OMG Document 91.12.1.

Perry DE. 1987. Software interconnection models. *Proceedings of the International Conference on Software Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 61–69.

Perry DE, Wolf AL. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* **17**(4):40–42.

Prieto-Diaz R, Neighbors J. 1986. Module interconnection languages. *Journal of Systems and Software* **6**(4):307–334.

Rich C, Waters R. 1988. The programmer's apprentice: A research overview. *IEEE Computer* **21**(11):10–25.

Rich C, Wills L. 1990. Recognizing a program's design: A graph parsing approach. *IEEE Software* **7**(1):82–89.

Rugaber S, Stirewalt K, Wills L. 1995. Detecting interleaving. *Proceedings of the International Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; pp. 265–274.

Shaw M, Deline R, Klein DV, Ross TL, Young DM, Zelesnik G. 1995. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering* **21**(4):314–335.

Soni D, Nord RL, Hofmeister C. 1995. Software architecture in industrial applications. *Proceedings of the International Conference on Software Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 196–207.

Tonella P, Antoniol G, Fiutem R, Merlo E. 1997. Variable precision reaching definitions analysis for software maintenance. *Euromicro Working Conference on Software Maintenance and Reengineering*; IEEE Computer Society Press: Los Alamitos CA.

Tonella P, Fiutem R, Antoniol G, Merlo E. 1996. Augmenting pattern-based architectural recovery with flow analysis: Mosaic—a case study. *Proceedings of the Working Conference on Reverse Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 198–207.

Wills L. 1992. Automated program recognition by graph Parsing. *Ph.D. Dissertation*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge MA.

Wong K, Tilley SR, Muller HA, Storey MD. 1995. Structural redocumentation: A case study. *IEEE Software* **11**(6):46–54.

Yeh AS, Harris DR, Reubenstein HB. 1995. Recovering abstract data types and object instances from a conventional procedural language. *Proceedings of the Working Conference on Reverse Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 227–236.

Yeh AS, Harris DR, Chase MP. 1997. Manipulating recovered software architecture views. *Proceedings of the International Conference on Software Engineering*; IEEE Computer Society Press: Los Alamitos CA; pp. 184–194.

Authors' biographies:

Roberto Fiutem received the Laurea degree in Electronic Engineering from the Politecnico of Milano, Italy in 1988. From 1989 he was a researcher at the Istituto per la Ricerca Scientifica e Tecnologica (IRST), Trento, Italy, where he was involved in artificial intelligence and software engineering research projects. He is now at Sodalia s.p.a., a telecom software industry in Trento, where he works on software engineering methodologies and tools, in particular in the object-oriented domain. His email address is: fiutem@sodalia.it

Giuliano Antoniol was born in Transacqua, Trento, Italy, on January 12, 1956. He received his doctoral degree in Electronic Engineering from the University of Padua in 1982. He is currently with IRST, a research institute in the north of Italy, where he is involved in several research and industrial projects. Results have been achieved in the architectural analysis of distributed software system and in C/C++ code analysis. His current research interests include software engineering, architectural recovery, reverse engineering, and distributed systems. His email address is: antoniol@itc.it

Paolo Tonella received his Laurea degree cum laude in Electronic Engineering from the University of Padua, Italy, in 1992, and his PhD degree in Software Engineering from the same University in 1999 with the thesis 'Code Analysis in Support to Software Maintenance'. Since 1994 he has been a full time researcher of the Software Engineering group at IRST (Institute for Scientific and Technological Research), Trento, Italy. He participated in several industrial and European Community projects on software analysis and testing. His current research interests include software engineering, reverse engineering, object oriented programming and code analysis. His email address is: tonella@itc.it

Ettore Merlo is an Associate Professor of Computer Science at the Ecole Polytechnique de Montreal. Previously he was the lead researcher of the software engineering group at CRIM (Computer Research Institute of Montreal). His research interests are in software analysis, software re-engineering, user interfaces, software maintenance, and artificial intelligence. He has collaborated with several organizations on projects in software re-engineering, clone detection, software quality assessment, and architectural reverse engineering. Ettore's Ph.D. is in Computer Science from McGill University in Montreal, and his Laurea degree summa cum laude is from the University of Turin in Italy. His email address is: merlo@rgl.polymtl.ca

